Guide to Local State

Mattox Beckman
<beckman@iit.edu>

May 10, 2006

1 Introduction

This document describes how state can be used to allow functions to remember things between functions calls. This will allow for some advanced techniques, such as counters, objects, and thunks.

1.1 Objectives

- Know the syntax in OCaml for specifying state operations.
- Understand how state and scope can be combined to create "local state".
- Be able to define multiple functions that share an encapsulated state.

2 OCaml State Syntax

State, in this context, means that a variable may change it's value during the execution of a program. Be careful here: we are not talking about the situation where multiple `let` statements define variables with the same name. In that case, there are simply two variables being defined that happen to have the same name. They actually exist in separate memory locations. The following code is an example of this:

```
# let x = 10 in
  let x = x + 1 in
    print_int x;;
- : int = 11
```

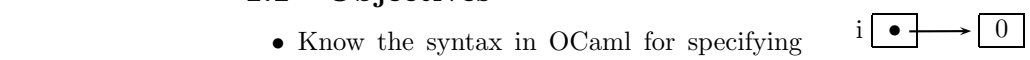
If we were to diagram this in memory, it would look like this:

In OCaml, you can declare a special type called a *reference* that is allowed to be changed. This is accomplished with the `ref` keyword.

```
# let i = ref 0;;
val i : int ref = contents = 0
```

This command allocates some memory on the heap, and initializes it with the integer `0`. The location is then assigned to `i`. Thus, `i` is a *reference* to the `0` on the heap.

In memory, we would diagram it as follows:



If you are familiar with Java, then the `x` variables from the first example correspond to the `int` type. They are called *primitives*. The `i` variable would correspond to the `Integer` type.

To access the value to which `i` refers (i.e., dereferencing), we use the prefix operator "`!`"

```
# i;;
- : int ref = {contents = 0}
# !i;;
- : int = 0
```

If you ask for `i` directly, you get the reference. If you ask for `!i`, then you get the value resulting from following the reference.

To update the contents of the reference, we use the `:=` operator.

```
# i := 30;;
- : unit = ()
# !i;;
- : int = 30
```

The memory diagram after this assignment would be like this:

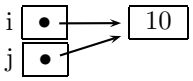
This reassignment is known as *destructive update*, because the old value ceases to exist once it is replaced by the new value.

2.1 Sharing State

One of the consequences of references is that multiple variables can refer to the same location in memory. Consider the following code:

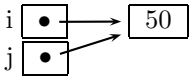
```
# let i = ref 10;;
val i : int ref =  {contents = 10}
# let j = i;;
val j : int ref =  {contents = 10}
# i := 50;;
- : unit = ()
# !i;;
- : int = 50
# !j;;
- : int = 50
```

The memory diagram at the point before the assignment is as follows:



Notice that we have two variables, but they point to the same location in the heap.

Now, after the update, we have



This is why changing `i` also changes `j`.

3 State and Equational Reasoning

Remember the rule of referential transparency. It says that if you have two expressions e_1 and e_2, and some external expression $C[\![]\!]$, then $C[\![e_1]\!] = C[\![e_2]\!]$.

For example, if we have

```
10 * (f(x) + f(x))
```

as an expression, then we can rewrite it as

```
10 * (2 * f(x))
```

since `f(x) + f(x) == 2 * f(x)`.

This is a powerful concept. It allows us to use the full power of mathematics (a science that has about 3,000 years more development than computer science) to help us reason about our programs. We can use this to prove properties of our programs, such as correctness, and this is the basis for code optimizations.

Unfortunately, things become a lot more complex when state is involved. Normally, you can take it for granted that

$$a + a = 2a$$

for any value a, but consider what happens if we have a function `counter` that returns a new integer, larger than the one returned previously.

```
# let ct = ref 0;;
 val ct : int ref = {contents = 0}
# let counter () =
    (ct := !ct + 1; !ct);;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
# counter ();;
- : int = 3
```

If we take `counter` as our a, then we get the following session:

```
# 2 * counter ();;
- : int = 8
# counter () + counter ();;
- : int = 11
```

Because of the reference, our function's value now depends on the changeable state of the `ct` variable. We have lost our ability to perform equational reasoning with this code, at least not without some far more complex mathematics.

The problem is that mathematical variables are timeless. Once we describe them and determine their value, then that value does not change during the evaluation of an expression. If we have state, then the value of a variable can change during the evaluation of an expression. Mathematical variables do not have this property.

While state greatly increases the complexity of our programs, there is one inescapable fact: our world is a stateful one. The amount of money in your bank account, the contents of your mailbox, the number of cars waiting at an intersection—these things are constantly changing, and state is the most natural model to describe them. Even the memory used to store variables is stateful. As a result, there are certain problems for which state will be the model to chose.

Currently, however, the trend in programming is to rely far too much on state to represent

things. This over reliance on state is bad practice, because it violates a basic tenet of design: don't use more complexity than what is needed to accomplish the task. This tenet can be seen in the decision to replace pointers with references in Java. Full-fledged pointers are far more powerful—and far more complex—than what is needed in all but a few specialized programming tasks, such as writing an operating system or a device driver. References have the capability we need most of the time, without the added complexity, and using them instead of pointers increases our productivity because the resulting software will be less complex, and thus easier to verify.

The conclusion of the sermon is this: use state if you must, but only if you must.

4 Local State

You are all familiar with using state to have two functions communicate with each other. One way this is done is by using global variables. These are also discouraged, because the programmer has little control over who can modify the variable, and later there is little indication in the program about where the variable itself is supposed to be used.

One technique of using state that does not have this problem is to restrict the state's accessibility to the definition of a function. Before we discuss that, it's best to review what you know about scope.

4.1 Review of Scope

Consider the following code. There are seven variables. Be sure you can explain the scopes of each of them.

```
1  let x = 10;;
2
3  let foo y = match y with
4   | 0,b -> let c = b * b in
5            let d = c * c in
6               b * c * d
7   | a,b -> map (fun z -> z + a + x) [a;b]
```

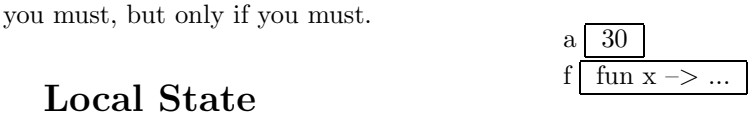
Here are the answers.

- `x` exists from line 2–7.
- `y` exists from line 3–7.
- `b` exists from line 4–6.
- `c` exists from line 5–6.
- `d` exists on line 6 only.
- `a` and `b` exist on line 7 only.
- `z` exists on line 7, in the body of the anonymous function.

Global declarations and functions Now consider the following function definition:

```
let a = 10 + 20
let f x = a + x
```

After the definition has been entered, the memory diagram might look like this:



Suppose `f` is called five times now. How many times will the "`10+20`" be computed? The answer is once. When `a` is defined, the value is computed. The function `f` uses the value five times, but this does not cause the computation to be redone.

Declarations from within a function Now consider this program:

```
let g y =
  let b = 10 + 20 in
     b + y
```

Once the definition is complete, the memory would look like this:

This `b` has not yet been created. That is part of the function's definition. If `g` is called five times, the `10 + 20` gets computed five times, because `b` is recreated each time.

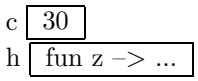
Note also that, unlike `a`, `b` is inaccessible to the rest of the program. It exists only within the scope of `g`.

Local declarations from outside a function It is possible to create a variable that is accessible only locally, but is only computed once. Consider this variation:

```
let h =
  let c = 10 + 20 in
    fun z -> c + z
```

Here, the variable `c` is defined as being `30`, and it's value is used in the body of the `let` expression. The result of the `let expression` is a function, which is then assigned to `h`.

The memory diagram might look like this:



Note well, however, that `c` is not accessible from the rest of the program. It was defined by the `let` expression, and existed only in the body of the `let`. After that, the name went out of scope.

The function `h` is able to use the value of `c`. But, unlike `g`, the function `h` does not recompute `c` each time it is called: the computation takes place before the function's body is defined, and the function body simply uses the result.

If we call `h` five times, `c` will still only be evaluated once. This behavior is a combination of `f`'s and `g`'s. The local variable is defined outside of the function body, so it is only defined once. But, because `c` is defined with a limited scope (via the local `let`), it is also not accessible from outside the function.

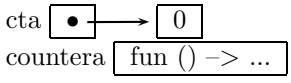
4.2 Scope and State

Interesting things happen when the local variable is a stateful one. Here are the three cases we discussed above, but with stateful variables.

Global declarations and functions

```
let cta = ref 0
let countera () =
  cta := !cta + 1;
  !cta
```

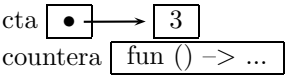
The resulting memory diagram would be like this:



In this variation, the variable `cta` is global, and can be accessed by anyone. If we call `countera` three times, we will have this session:

```
# countera ();;
- : int = 1
# countera ();;
- : int = 2
# countera ();;
- : int = 3
```

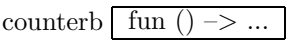
The memory diagram will then look like this:



Declarations from within a function Now see what happens if we declare the state inside the body of the function.

```
let counterb () =
  let ctb = ref 0 in
    ctb := !ctb + 1;
    !ctb
```

The memory diagram will be analogous to our function `g`.



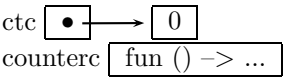
The `ctb` variable is recreated each time the function is called. As a result, the counter never changes.

```
# counterb ();;
- : int = 1
# counterb ();;
- : int = 1
# counterb ();;
- : int = 1
```

Local declarations from outside a function Here is where things get interesting. Suppose we define a state using the technique from function `h`.

```
let counterc =
  let ctc = ref 0 in
    fun () -> ctc := !ctc + 1;
              !ctc
```

The memory diagram will look like this:



Like the `h` example, the variable `ctc` will not be visible outside of the function. But, it will not be recomputed each time, either. The variable `ctc` was created first, and its definition used to build the function. This gives us the following session:

```
# counterc ();;
- : int = 1
# counterc ();;
- : int = 2
```

```
# counterc ();;
- : int = 3
```

4.3 Multiple Functions

It is also possible to have multiple functions share a common state. Consider this session:

```
# let (counter, reset) =
    let ct = ref 0 in
      (fun () -> ct := !ct + 1; !ct),
      (fun nv -> ct := nv);;
val counter : unit -> int = <fun>
val reset : int -> unit = <fun>
# counter ();;
- : int = 1
# reset 5;;
- : unit = ()
# counter ();;
- : int = 6
```

We can do this because functions are first class. We can put them into a tuple, and return the tuple. This has the effect of allowing multiple functions to be defined at once. Both `counter` and `reset` were defined within the scope of `ct`, our local state. As a result, both have access to it.

This should remind you of object oriented programming.

5 Conclusion

So, we have a technique for encapsulating a state within a function. This allows us to have functions that remember things. This can be used to have functions cache their results, or have functions which change their values.

Of course, these are no longer functions in the mathematical sense, but used properly, they can model real-world behaviors very effectively.